Empirical results
of long-lived renaming algorithms

Sandra Moral
José Luis Balcázar

Empirical results on long-lived renaming algorithms*

Sandra Moral     José Luis Balcazar

June 1997

Abstract

The Long-lived Renaming problem is an important subject in Distributed Algorithms. The Renaming problem consists in providing processes with new names from a hopefully smaller name space. In the long-lived version, only a fraction of the processes request a new name and later on abandon it and this name could be acquired by another processor. A bound is assumed on the number of processors holding simultaneously a name from the smaller name space. Only recently wait-free fast algorithms for long-lived renaming on asynchronous and shared memory distributed systems have been found and analyzed, and they had not been actually implemented. We describe some empirical considerations complementing the formal static analysis, and assess how useful each of them could be in a real application, as a result of the study of their implementations on a running asynchronous distributed system.

Keywords

Long-lived-renaming problem, Distributed System implementation, Shared Memory, Asynchronous Systems, FILTER, SPLIT

1 Introduction

The long-lived renaming problem is an important problem on distributed systems which perform some work with a subset of processes. It is particularly applicable when the efficiency of the algorithm where the task performed by the processes depends on the size of the name space, as is the case for some algorithms. Renaming algorithms see this as a sequence one-shot problem: some processes request and acquire new names from the smaller name space. Long-lived renaming is an on-going problem: process request and acquire new names, and eventually release them for use by another processor. This view is important if the subset of processes allocated to do the task is dynamic, as will be the case in many realistic settings (if not otherwise, to react to faults).

The long-lived renaming problem could be defined as follows: The distributed system is composed by \( n \) processes. Each process has an identifier from the set

---

*Partially supported by ESPRIT Long Term Research Project no. 20244 (ALCOM-IT) and by the Spanish DGICYT through project PB55-0737 (KOALA).
\( \{0, \ldots, S - 1\} \), and only some of them, never more than \( k \) simultaneously a new identifier \( (k \ll n) \); the new identifier should be from a smaller set \( \{0, \ldots, D - 1\} \) \( (D \ll S) \). Thus, once \( k \) names are acquired, no other processor will request one until at least one of the used names is released by the processor holding it. The new name acquired could be different from the one released.

Of course, efficiency of the algorithms for this problem is rather critical, as their are intended to reduce the size of the name space for the sake of efficiency of the underlying computation, and their use must be offset by the gains obtained. Besides an efficiency comparable to these gains, two other properties are desirable for these algorithms: they should be fast and wait-free.

A protocol is fast if its time complexity is polynomial in the number of processes acquiring and releasing names \( (k) \) and is independent from the total number of processes \( (n) \) and the size of the source name space \( (S) \). A protocol is wait-free if the processes do not have dependencies among them that can influence on their speed. Thus, slow (or failing) processes will not force faster processes to slow down. These two properties are very important when processes can fail, when their relative speed is different or when the number of processes in the global system is very large.

To obtain a wait-free protocol, one must be careful with the use of mutual exclusion barriers in accessing variables. Without barriers a process never waits actively to another process. One of the main features of one of the solutions implemented is that the use of a mutual exclusion mechanism is careful enough to still obtain a wait-free protocol.

The first solution of these problem can be found in [2]; after it, some more solutions have been obtained such as those described in [7] and [3]. There are also some solutions for others kinds of environments, such as communication by message passing or one-time acquisition of names (see [5], [1], [9]).

Within this paper, we focus on two fast, wait-free algorithms from [3]: FILTER and SPLIT. This work describes their implementation, testing, and mutual comparison of these two long-lived renaming algorithms. After this introduction, we introduce in section 2 the first algorithm, FILTER, and in section 3 the second one, SPLIT. Concluding remarks and future work appear in the last section.

2 First Long-lived renaming problem solution:
FILTER

The long-lived renaming algorithm presented in this section, FILTER, is from [3], and is based on the mutual exclusion blocks of Peterson and Fischer [6]. In FILTER, these blocks are arranged in a tree called Mutual Exclusion Tournament Tree, and there is such a tree for each destination name. A process acquires a name \( m \) when it reaches the root of the tree \( T_m \). Processes start at a leaf and try to climb up the tree. The risk of a process waiting for another one is avoided by having each process try several trees in parallel until it reaches a root in one of them.

We call \( N_p \) to the set of names that process \( p \) can try to acquire, i.e. the set of trees in which the process will play his game. This set is chosen in such a way that the intersection between sets corresponding to different processes is very small.
The sets are generated using hashing techniques, based on families of finite sets with small intersections [8]. The cardinality of the sets used are a bit bigger than necessary, because this improves the efficiency.

The following section shows (in detail) the method used by a process to advance a level in a tree. In section 2.2 we present the hashing method used to find the name set \( N_p \). The section 2.3 contains the acquire algorithm, and in section 2.4 there is the theoretical time complexity. And, to conclude, the last section discusses some empirical results obtained after testing the algorithms.

2.1 Steps needed to achieve the root of a Mutual Exclusion Tournament Tree.

When a process \( p \) needs a name, it tries to reach the root of a mutual exclusion tournament tree competing with other processes; it should do the following:

I. Enter the tree through the appropriate leaf, which depends on the old process identifier.

II. Compete with other processes (if any) to achieve the next level:

   A. Register its presence and the direction used to arrive (using procedure Enter) in the reached block.

   B. Check whether it can continue climbing up the tree (using function Check).

III. Repeat point II until it is on the root.

A process can be stopped by any other process ascending in the same tree (for example winning the point II.B before the other). This waiting time is not an active waiting because the stopped process can try another tree in its name set \( (N_p) \). Furthermore, the size of this set is chosen in such a way that it guarantees that a process always reaches a root. See [3] for a proof.

2.1.1 Detailing Enter, Check and Release

To build the functions cited on the last Section, we need a shared variable defined as follows:

\( R \) is an array of two booleans \( \{0, 1\} \) set initially by \( no\_def \). This array contains the information about the direction of the process entered.

\[
\text{Enter}(\text{ME:block}; d:\text{direction}): \\
\text{if} \ (R[1-b] = no\_def) \ \text{then} \\
\quad R[b] := true; \\
\text{else} \\
\quad R[b] := b \oplus R[1-b]; \\
\text{end if} \\
\text{if} \ (R[1-b] \neq no\_def) \ \text{then} \\
\quad R[b] := b \oplus R[1-b]; \\
\text{end if}
\]
if (current_level = log S) and
    (Check block of the current level)=true)
then
    /* root reached, name acquired */
    return np(i) name
end if;
end for;
until it gets a name
end repeat;

2.4 Theoretical Results

The previous algorithm has a time complexity which depends partially on the size of the source name space $S$. We describe the time complexity in the table below.

<table>
<thead>
<tr>
<th>Source Name Space ($S$)</th>
<th>Destination Name de Space ($D$)</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \leq 3k - 1$</td>
<td>$2k^4$</td>
<td>$O(k^3)$</td>
</tr>
<tr>
<td>$2k^4$</td>
<td>$72k^2$</td>
<td>$O(k \log k)$</td>
</tr>
<tr>
<td>$S$ polynomial in $k$</td>
<td>$O(k^4)$</td>
<td>$O(k \log k)$</td>
</tr>
</tbody>
</table>

Table 2.4 - Size of Name Spaces vs Time Complexity.

To verify that all process always manage to acquire a name we offer to each process a larger set of possible names. The set is constructed in such a way that a process advances a level in every round, so a process always has a name where it can advance a level to the root.

2.5 Empirical Results

The implementation and testing of FILTER has been done in a Silicon PowerArray, an asynchronous distributed system with shared memory communication. The language used is C with appropriate compiler directives for parallelization; specifically, the shared variables needed to implement the tournament trees are declared using the adequate pragma directives.

Processes use shared memory as a model of communication and are executed in an asynchronous way. The atomic operations used are just atomic read and atomic write. It must be pointed out that the availability of more powerful atomic operations such as set-first-zero, bounded-decrement, and others, allows for simpler and/or more efficient algorithms, and that actually some of these operations were available in the system used.

Several tests were run, towards a study of the worst cases of the algorithm. We realize that the main feature of the protocol is the hashing function used to select the set of names $NP$. This function is not always available on all distributed systems, since the equations given in 2.2 must be adhered to. So, we focus our results on this fact, studing the hashing function in detail. We can conclude that only when the value of the number of processes involved on the renaming $k$ is very small comparing it with the global number of processes $n$, the hashing method is applicable. For example, in
a distributed system with 45000 processes the method only fits in the restrictions if the subset of processes ($k$) is around one hundred. This huge contrast is shown in Figure 2.5-1 Once a maximum number of processes $k$ is fixed we can study the other hashing parameters to obtain the smallest size of the destination name space $D$. The values obtained can be seen in Figure 2.5-2). These values show that to get an important reduction of the name space, $n$ should be huge and $k$ rather limited. For example, when the system has identifiers from $\{1, \ldots, 3000\}$ and only 3 processes are trying to get a new identifier, the new identifier will be in the range $\{1..500\}$.

There are other interesting consequences of this study: the degree of the polynomial, characterizing the maximum number of collisions, can be maintained rather low with an affordable increase in the value of the prime number $z$; and, above all, even though the small number of really independent processors in the target machine makes this prediction risky, the figures obtained from the simulation certainly suggest that, as the theoretical study predicts, the running time of the algorithm is small enough to be negligible in a practical setting (it compares with the time needed by the system to just set up the processes), even for huge initial name spaces.
3 Second Long-lived renaming problem solution: SPLIT

The SPLIT algorithm gives another fast and wait-free solution for the long-lived renaming problem. The algorithm in detail is in 3.3; it can reduce the name space to $k(k+1)/2$ with a time complexity $O(k^3)$. In general, this algorithm can reduce the source name space (with size $S$) to a destination name space $3^{k+1}$ with a time complexity $O(k)$. In the following subsection we show the method used by the processes to acquire a name without mutual exclusion mechanisms. At the last subsection we compare the results obtained.

3.1 Competing to acquire a name

This algorithm uses a splitters tree to give names to the process who try to get a new identifier. Processes go down to the tree until they reach a leaf, where they can calculate their new identifier. Each block that is not a leaf redistributes the processes reaching it among its subtrees, so that when a process reaches a block it continues down the tree; the blocks ensure that no two processes follow exactly the same path.

In this version we use a ternary tree of building blocks, so every block can distribute process into three directions (one for each subtree). The routing function of every block must guarantee that the directions given to the processes are not always the same. The final name can be obtained applying a function over the path constructed going down by the tree.

3.1.1 Going through a building block

After describing the global idea of the algorithm, we focus now on the specific method used by a building block when some process arrives. The function $Enter(B,p)$ guides the process $p$ when it crosses block $B$, returning one of the three directions $-1, 0, 1$.

When the process $p$ will not use anymore the name obtained, it should release each block visited $B$ with the function $Release(B,p)$. These functions are defined in such a way that many processes can reach the same block at the same moment without any mechanism of mutual exclusion but using some mechanism to execute some parts atomically. The algorithms of these functions are presented below.

Local variables used by a process:

- $selected\_dir \in \{-1, 1\}$ informs to the other processes of a direction that can be selected (it is a safe direction).
- $am\_I? \in \{\text{true, false}\}$ it indicates whether the process is the last one which has entered that block.

Shared variables used by a process:

- $last\_proc \in \{0, ..., S - 1\}$ it shows the last process entered.
- $last\_dir \in \{\text{no\_def, -1, 1}\}$ initialized to 1, represents the last direction given by the block to a process.
• safe_dir ∈ \{-1, 1\} initialized to 1, informs to the other processes that they can choose this direction because not all processes that have crossed the block had chosen it.

Enter (B is a block, p is a process) return direction:

1: last_proc = p;
2: selected_dir := last_dir;
3: if (selected_dir = no_def) then
   selected_dir := safe_dir;
   end if
4: last_dir := - selected_dir;
   selected_dir := last_dir;
5: am_I? := (last_proc = p);
6: if (am_I?) then
   safe_dir := - selected_dir;
   end if
7: if (last_proc = p) then
   return (selected_dir);
else
   return(0);
end if
8: Work Section Block

Release (B is a block, p is a process)

9: if (last_proc = p) then
10:   last_dir := selected_dir;
   end if
11: if (not am_I?) then
   last_dir := no_def;
   end if

3.2 What happens when two processes reach a block simultaneously

The main function of the splitters tree is to guide the process to different directions. So, if l processes reach a block, the functions defined in previous sections guarantee the following:

• if they work synchronously, they will obtain l – 1 directions with value 0 and only the last one will make true the step 7.

• if they work asynchronously, the function still will be fair because not all processes will take direction 1. Processes will take direction –1 when:
  - some processes had entered into the block while it was on step 2,3 or 4.
  - some processes release the block initializing the last_dir with no_def value.

The following property should be an invariant for all implementations of the presented functions: “the number of processes with the same output direction from a block must always be smaller than the number of processes that use it”.
3.3 Algorithm SPLIT

procedure GetName \( p \in [0, \ldots, S - 1] \)

\[\begin{array}{l}
\text{var}\\
\quad RN \text{ array of building blocks;}\\
\quad s \text{ string of } \{0, 1, -1\};\\
\quad level \in \{1, \ldots, k - 1\};\\
\quad e \text{ string of } \{0, 1, -1\};\\
\quad s_i \in \{0, \ldots, D - 1\};
\end{array}\]

\[s := \text{empty string};\]
\(* \text{ Going down the tree *})\]
for all level \( \in [1 \ldots k - 1] \) do
\[\begin{array}{l}
\quad e := \text{Enter} \left(RN[s], p\right);\\
\quad s := (S \ e);\\
\end{array}\]
end for
\[s_i := \sum_{i=1}^{k-1}(1 + s[i])3^i - 1;\]

return \( s_i \);

end

procedure ReleaseName \( p \ \text{integer} \in \{0, \ldots, S - 1\}\)

\[\begin{array}{l}
\text{var}\\
\quad RN \text{ array of building blocks;}\\
\quad s \text{ string of } \{0, 1, -1\};\\
\quad level \in \{1, \ldots, k - 1\};
\end{array}\]

for all level \( \in [k - 1 \ldots 1] \) do
Release\(RN[s], p\);
\[s := s[1 : |s| - 1];\]
end for

end

3.4 Empirical Results

Test runs of the implementation confirm the impression, that can be obtained from the understanding of the algorithm, that in this protocol is the asynchrony inherent to the system the source of the scattering of the process among the available names in the smaller name space: the name obtained depends directly of the asynchrony and of the concurrence of processes in the blocks. It is clear from the results that the obtained name is indeed independent of the original name of the process, and that no regularity at all is apparent in the outcoming distribution of names. Indeed, the quantity of processes that have traversed previously a given block, and even more the coincidence of processes at blocks, depend to such a fine grain on their relative speeds that the obtained name distribution lacks any visible regularity. On the other hand, this is not true of FILTER, where a precomputation of the name sets and potential collisions gives an a priori perspective of the name distributions one can expect.
4 Conclusions and future work

In this article we have presented two long-lived renaming algorithms: \textsc{filter} and \textsc{split}; both are from [3]. There did not exist an implementation validating in practice these theoretical constructions, which relied on rather deep mathematical facts such as the properties of polynomials in finite fields [8]. We have described the experimental facts obtained through implementations on a commercial distributed shared-memory system. The empirical observations corroborate most theoretical observations, the main differences being, on the one hand, that the values obtained for the destination name spaces may be somewhat high, and, in practice, much higher than what seems really necessary. So to say, the worst case that the theory prevents against looks extremely unlikely. A natural (but probably very difficult) alley for further research would be to undertake a formal analysis of the average size necessary for the destination name space.

Although the \textsc{filter} solution is more sophisticated than \textsc{split}, \textsc{split} manages to give a solution for all distributed systems, in a rather simpler way, whereas \textsc{filter} requires a considerably large destination name space even for just a handful of processes holding a destination name simultaneously. The advantage of \textsc{split} lies essentially in its independence of the old identifier, which makes it rare to find any sort of critical execution. It will adapt better to systems where the number of processors requesting a new name may vary. On the other hand, the more varied is the set of requesting processors, the more invalid tests are performed by \textsc{filter}, whereas \textsc{split} does not seem to be that affected.

For all these reasons we think that future work can be centered on the improvement of the \textsc{filter} solution. The main feature to optimize should be the hashing method (see 2.2). First, a somewhat smaller destination name space can be used if a slightly less efficient algorithm is accepted (climbing up at least one tree per round, instead of at least half of them as we do in the implemented version): since $|N_p| > d(k - 1)$ can be shown to suffice, using $D = z(d(k - 1) + 1)$ instead of $D = z2d(k - 1)$; thus $|N_p| = d(k - 1) + 1$, guaranteeing that at least one tree will be advanced, instead of $|N_p| = 2d(k - 1)$ to do it in half of them. We consider this reasonable to try since our experimentation indicates that running time is not to be expected (for now) to be a real bottleneck.

On the other hand, the slowdown might be reduced: we think that is possible to bring \textsc{filter} down to more desirable times in the critical cases, if a process $p$ which has achieved a root, releases immediately all the trees it is not using, instead of waiting to release all of them at once. In that case other processes could get the names locked by a process which has secured already its own tree. However, both a formal analysis and a reliable experimental analysis of these proposals seem hard to do. Finally, regarding \textsc{split}, the natural continuation would be to experimentally measure the extent to which the algorithm can be improved by using cuaternary trees.

Previous studies about this problem did not give any proof of fairness of the presented solutions. Fairness is always very appreciated property and not always achieved. We consider a protocol \textit{fair} if all processes are able to acquire a name if we work on an infinite execution of the protocol. Actually we are working on the fairness of the method using the methodology and rules of by Hoare [4].
References


LSI-97-1-R “On the Number of Descendants and Ascendants in Random Search Trees”, Conrado Martínez and Helmut Prodinger.


LSI-97-3-R “Solving Incidence and Tangency Constraints in 2D”, Núria Mata.

LSI-97-4-R “Designer: A Tool to Design and Model Workflows”, Camilo Ocampo, Pere Botella.


LSI-97-6-R “The Extreme Vertices Model (EVM) for Orthogonal Polyhedra”, A. Aguilera and D. Ayala.


LSI-97-8-R “Randomized Binary Search Trees”, Conrado Martínez and Salvador Roura.


LSI-97-12-R “Single-Pushout Hypergraph Rewriting through Free Completions”, Ricardo Alberich, Francesc Rosselló, and Gabriel Valiente.


LSI-97-14-R “NoFun: A Notation to State Non-Functional Specifications at the Product Level”, Xavier Franch.


LSI-97-16-R “A Dichotomy Theorem for Learning Quantified Boolean Formulas”, Víctor Dalmau.

LSI-97-17-R “Rule generation from real data: GAR meets LINNEO+”, David Riaño and Ulises Cortés.

LSI-97-19-R "Especificación de restricciones de integridad en el sistema ROSES" (written in Spanish), Maria Amélia Pacheco e Silva and Maria Ribera Sancho i Samsó.


LSI-97-22-R "Refining Logical Characterizations of Advice Complexity Classes", Albert Atserias and José L. Balcázar.


LSI-97-26-R "The (Parallel) Approximability of Non-Boolean Satisfiability Problems and Restricted Integer Programming", Maria Serna, Luca Trevisan and Fatos Xhafa.

LSI-97-27-R "Mean Field Theory of Fluid Neural Networks", Jordi Delgado and Ricard V. Solé.

LSI-97-28-R "The Structure of Logarithmic Advice Complexity Classes", José L. Balcázar and Montserrat Hermo.

LSI-97-29-R "Structuring the Process of Integrity Maintenance (Extended Version)", Enric Mayol and Ernest Teniente.


LSI-97-31-R "Parallel Dictionaries with Local Rules on AVL and Brother Trees", Joaquim Gabarró and Xavier Meseguer.


LSI-97-34-R "A Short Note on Non-Symmetric Semidefinite Programming", Fatos Xhafa.


Hardcopies of reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Campus Nord, Mòdul C6
Jordi Girona Salgado, 1-3
08034 Barcelona, Spain
scretisiolsi.upc.es

See also the Department WWW pages, http://www-1si.upc.es/www/